A SCALABLE ALGORITHM FOR COMPILER-PLACED STAGGERED CHECKPOINTING

Alison N. Norman
Department of Computer Science
The University of Texas at Austin
Austin, Texas, USA
ans@cs.utexas.edu

Calvin Lin
Department of Computer Science
The University of Texas at Austin
Austin, Texas, USA
lin@cs.utexas.edu

ABSTRACT
To make progress in the face of possible system failures, long-running parallel applications often checkpoint, or save their state, so they can resume execution. Many current checkpointing techniques require user input, impose run-time performance penalties, or result in all processes checkpointing synchronously which leads to network and file system contention, again resulting in significant performance penalty. This paper presents an algorithm to perform compiler-placed staggered checkpointing, where process checkpoints are placed at compile-time at different places in the application text, thereby reducing contention for the network and file system.

Identifying staggered checkpoints is algorithmically challenging since the number of possible solutions is enormous—it grows as L^P, where L is the number of possible checkpoint locations and P is the number of processes—and the number of desirable solutions is relatively small. But our algorithm successfully places staggered checkpoints in parallel applications that use tens of thousands of processes. On benchmarks, our algorithm generates and places checkpoints that improve performance by an average of 35% over the current technique.

KEY WORDS
Checkpointing, Supercomputing, Parallel Computing, Compilers and Run Time Support

1 Introduction

Supercomputers are large-scale systems often composed of many thousands of processors. These machines support long-running parallel applications that can use tens of thousands of processes. Unfortunately, these applications might encounter failures before they complete, due to problems with the application itself, system software errors, or hardware failures [1–3]. To mitigate the effect of these failures, each process can periodically save its state in a *checkpoint* so that the application can restart execution after a failure. The checkpoint and recovery process is greatly simplified if the set of checkpoints can be guaranteed to represent a *consistent state* [4], which means that the state could have existed during the execution of the application.

Typically, programmers manually choose where the application checkpoints by placing a checkpoint call in the application source code, ensuring a consistent saved state by synchronizing all processes before and after the checkpoints. This technique is fairly straightforward to implement. Unfortunately, as the number of processes grows, these *synchronous checkpoints* suffer from significant overhead due to network and file system contention [5].

This paper presents an alternative to programmer-placed checkpoints that scales to applications using tens of thousands of processes. In particular, we present an algorithm that allows compilers to place *staggered checkpoints*, where the process checkpoint calls are placed at different locations in the application text while guaranteeing that the saved checkpoints form a consistent state. Our solution assumes that spacing process checkpoint calls in the application text results in a corresponding temporal spacing during execution—creating a separation in time of each process's checkpoint and reducing network and file system contention and thus checkpoint overhead. Our algorithm guarantees the consistency of the checkpoint without: 1) dynamic coordination among processes, 2) message logging, or 3) user input. Our solution also allows processes to save checkpoints during inter-process communication.

It is difficult to place consistent staggered checkpoints in an application because the number of possible solutions grows as L^P, where L is the number of possible *checkpoint locations*, or sites in the application text where a process checkpoint call may be placed, and P is the number of processes, and the number of valid solutions is minuscule. For example, for the NAS Parallel Benchmark `BT` [6] using just 16 processes, 38^{16} possible solutions exist but only 191 of those are valid. Desirable solutions—those that both save a consistent state and are sufficiently staggered to reduce contention—are even more rare. Thus, any algorithm that arbitrarily reduces the search space is unlikely to find desirable solutions. Our algorithm uses informed techniques to reduce the search space—enabling it to reduce the search space for `BT` using 1,024 processes by 1,594 orders of magnitude before it begins generating solutions.

We also introduce a static metric for evaluating the desirability of a solution. This metric enables the algorithm to quickly identify solutions that reduce network and file system contention. We show that using this metric, the solutions chosen by our algorithm achieve a 24% decrease in checkpoint time when the checkpoints can be staggered over a three minute window. When given a fifteen minute window, checkpoint time is decreased by 44%.

2 Related Work

Network and file system contention can be reduced in three ways: (1) processes can save their checkpoints to some device other than the global file system, (2) checkpoint size can be reduced, and (3) the number of simultaneous checkpoints can be reduced.

The first approach often saves checkpoints to the local disk of another machine [7–9], but many supercomputers do not provide this form of storage. Alternatively, checkpoints can be saved to another node's main memory [9], but that is not suitable for memory-bound applications.

The second approach reduces the checkpoint size [10], but for large problem sizes, such solutions still incur significant contention.

Staggered checkpointing falls into the third category, where our work is distinguished by its scalability. Previous staggered checkpointing methods focused on systems with just a small number of processes [11, 12]. Other protocols reduce the number of simultaneously writing processes by partitioning processes and then performing coordinated checkpointing within the partitions [13, 14]; these protocols suffer from dynamic coordination and require message logging for communication between partitions. In message logging, processes write their messages to a log so that, if necessary the messages can be replayed during recovery to create a consistent state. Checkpointing techniques that use message logging require that the application has enough unused memory on the local hardware to store message logs, a requirement that is not met by many applications [15]. Another protocol [16] allows for uncoordinated checkpointing, which reduces contention for the network and file system but also requires message logging.

Previous compiler-assisted checkpointing work includes techniques that stagger checkpoints across communication-free zones [17, 18], but many applications do not have sufficiently large communication-free zones to reduce contention for large problem sizes. Another line of work [19, 20] uses compilers to assist in identifying valid recovery lines, but this work does not attempt to stagger the checkpoints to reduce contention.

3 Background

In this section, we define terms and concepts that we need to explain our algorithm.

A *recovery line* is a set of checkpoints, one checkpoint per process. A *valid recovery line* represents a consistent state, or a state that could have existed in the execution of the program [21], so it disallows the save of a message *receive* if the corresponding *send* of the message is not also saved. By ensuring that data shared amongst processes is

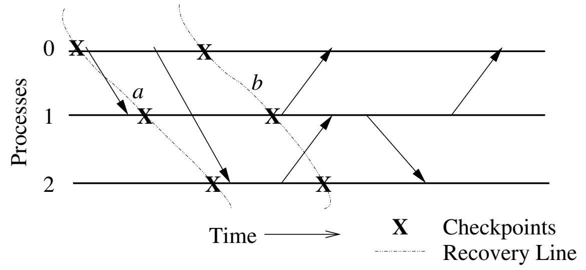

Figure 1. Examples of invalid (a) and valid (b) recovery lines. The arrows represent message sends. The source is the sending process and the target is the receiving process. The **X**'s mark checkpoint locations.

the same data everywhere, valid recovery lines save a consistent state even in the face of non-determinism in applications. For example, recovery line a in Figure 1 is invalid because it saves a message as received on process 1 while process 0 does not save the corresponding send. Thus, the state represented by a could not exist in an actual execution of the system. By contrast, recovery line b is valid because it could have existed in a possible execution.

Our algorithm specifies a recovery line as set of $\{$*process, checkpoint location*$\}$ pairs, with one pair for each process. Since our algorithm builds recovery lines incrementally, we define a *partial recovery line* to be a set of $\{$*process, checkpoint location*$\}$ pairs for a subset of the processes in the system; a *partial valid recovery line* is a partial recovery line that could be a subset of a valid recovery line.

Valid recovery lines can differ in the number of processes that checkpoint at approximately the same time, and thus they differ in terms of the network and file system contention that they induce. Thus, we refer to one recovery line as being *better* than another if it reduces that contention and so has lower checkpoint overhead.

4 Our Solution

Our solution strives to create *useful* recovery lines, which are lines that are both valid and consist of process checkpoint calls that are sufficiently staggered to reduce network and file system contention and thus checkpoint overhead. Our solution is applicable to either message passing or shared memory programs, including those using non-deterministic communication. Non-deterministic communication causes our algorithm to identify more conservative communication dependences.

Our solution has three phases: the first identifies all communication using a static analysis, the second determines inter-process dependences by statically creating vector clocks[1], and the third generates recovery lines. We focus on the last phase since it represents our most novel and important contributions, while the other two phases are relatively straightforward [22].

[1] Vector clocks were developed independently by prior researchers.

```
identify communication
divide application into phases (Sec 4.2.2)

for each phase
  perform control-dependence analysis to
  determine which processes execute each
  communication call and checkpoint location

  identify process dependences
  generate clumps (Sec 4.2.3)
  form clumpsets (Sec 4.2.3)

  merge any checkpoint locations not
    separated by a minimum number of local
    operations (Sec 4.2.2)

  for each clump set
    generate_lines() (Sec 4.2.1)
```

Figure 2. Pseudocode for Our Algorithm, Part 1

We restrict checkpoint locations to coincide with *dependence-generating events,* which are typically communication and synchronization operations. In principle, the checkpoint locations can be adjusted relative to these events to both increase the separation of the checkpoints and reduce the amount of checkpoint data, but we leave such considerations to future work.

The algorithm that generates recovery lines is designed to eliminate redundant work and reduce the search space. To place this work in context, we first briefly present the obvious depth-first search algorithm.

4.1 Brute Force Algorithm

An obvious algorithm uses exhaustive depth-first search (DFS) to combine {*process, checkpoint location*} pairs into recovery lines. It checks the validity of each line after each new {*process, checkpoint location*} pair and backtracks when a recovery line is found to be invalid. This algorithm, which has an executions complexity of $O(P^2 * L^P)$, where P is the number of processes and L is the number of checkpoint locations, becomes infeasible for even small values of P. Moreover, it performs much redundant work, repeatedly checking pair combinations already found to be invalid.

4.2 Our Algorithm

We now describe our algorithm, beginning with our foundation algorithm, which eliminates portions of the search space that contain no valid recovery lines. Then we present techniques that reduce the number of considered checkpoint locations, L. Finally, we describe two heuristics: the first reduces the number of considered processes, P, and the second preferentially produces useful recovery lines. The resulting algorithm is outlined in Figures 2 and 3.

Our solution assumes that applications are written in

```
generate_lines()
  divide clumps into partitions

  while (partition_size < # of clumps in set)
    combine partitions into sets

    for (each partition combination)
      if(initial clump/location pair is valid)
        merge partition combination into
          partial valid recovery line
        add new partial valid recovery
          line to set of recovery lines

        if(# of recovery lines >
           line check threshold) (Sec 4.2.7)

         for (each recovery line)
          if(line is valid)
            add line to valid set

            if(set of valid lines >
               pruning threshold) (Sec 4.2.7)
              evaluate valid lines using
                branch and bound policy with
                Wide-And-Flat (Sec 4.2.6)
                keeping the number of lines
                set by lines to keep
                (Sec 4.2.7)

      if(initial pair or line invalid) &&
         (# of recovery lines >
          line check threshold))
        use strictly increasing nature of
         dependences to prune (Sec 4.2.4)
```

Figure 3. Pseudocode for Our Algorithm, Part 2

the Single Process Multiple Data (SPMD) model and that communication is explicit at compile time.

4.2.1 Foundation Algorithm

The foundation algorithm is a constraint-satisfaction solution synthesis algorithm[2]that builds valid recovery lines by combining {*process, checkpoint location*} pairs into larger partial valid recovery lines until complete valid recovery lines are generated. Our algorithm begins by placing processes into partitions of size k. For each partition, the algorithm generates all partial valid recovery lines. It then merges j partitions to create a new partition and combines the partial valid recovery lines from each source partition to form new recovery lines, storing the valid ones. The algorithm repeats these steps until all processes are in the same partition and the generated valid recovery lines are complete. This technique eliminates redundant comparisons of invalid {*process, checkpoint location*} pair combinations by immediately pruning each invalid combination

[2]It is based on the basic Essex solution synthesis algorithm [23].

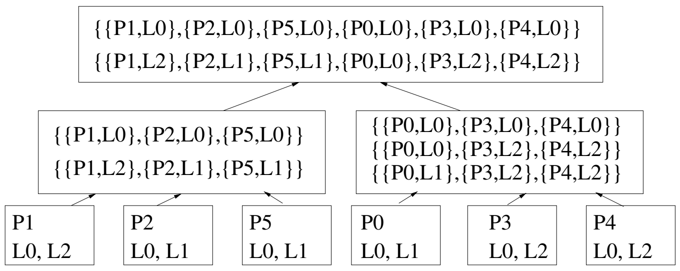
{{P1,L0},{P2,L0},{P5,L0},{P0,L0},{P3,L0},{P4,L0}}
{{P1,L2},{P2,L1},{P5,L1},{P0,L0},{P3,L2},{P4,L2}}
{{P1,L0},{P2,L0},{P5,L0}}
{{P1,L2},{P2,L1},{P5,L1}}
{{P0,L0},{P3,L0},{P4,L0}}
{{P0,L0},{P3,L2},{P4,L2}}
{{P0,L1},{P3,L2},{P4,L2}}
P1 L0, L2
P2 L0, L1
P5 L0, L1
P0 L0, L1
P3 L0, L2
P4 L0, L2

Figure 4. An example of our foundation algorithm, where $k = 3$ and $j = 2$. At the base of the diagram, each block represents a process and that process's checkpoint locations. In the top levels, each line in the merged partitions represents either a partial or complete valid recovery line.

from consideration. Figure 4 shows an example of how the foundation algorithm proceeds. We found the algorithm to be fastest and generate the best results for $k = 2$ and $j = 2$.

The execution complexity is $O(log(P) * L^{P-2})$.

4.2.2 Reducing L

To reduce the $O(L^P)$ sized search space, we reduce the value of L by dividing the application into *phases*, which are application segments delimited by events that synchronize all processes, such as barriers or collective communication. Valid recovery lines cannot straddle such events, so our algorithm searches within each phase. Furthermore, because the goal is to spread the checkpoints apart in time, the algorithm further reduces L by considering the set of well-separated locations within a phase. dditionally, in some applications so few *local operations*, or operations that execute entirely on the local machine, separate potential checkpoint locations that these locations are in close temporal proximity; separating process checkpoint locations across these events would be unlikely to reduce contention. The algorithm therefore considers any checkpoint locations that are not separated by some minimum amount of local operations to be a single location.

The execution complexity of the algorithm remains the same as that of the foundation algorithm $O(log(P) * L^{P-2})$ and the algorithm must now execute once per phase, but the value of L is now significantly reduced.

4.2.3 Reducing P

The number of checkpoint locations in a given phase is typically much smaller than the number of processes, so multiple processes must write their checkpoints at the same checkpoint location. If the processes that write their checkpoints together also communicate with each other, then the number of invalid recovery lines will be reduced since communication generates recovery line constraints. To exploit this property, our algorithm generates *clumps*, which are sets of processes that communicate between two consecutive checkpoint locations.The algorithm considers each clump as a single entity when it generates a recovery line, and so every process in a clump writes its checkpoint at the same checkpoint location.

The clumps are then further assigned into *clump sets*. A clump set is a group of clumps where each process is represented exactly once, such that a recovery line formed using every clump in a clump set is complete. Our recovery line algorithm is then modified to: 1) execute once per clump set and 2) place each of the set's member clumps in a recovery line rather than each process.

These techniques reduce execution complexity to $O(log(C) * L^{C-2})$, where C is the number of clumps in a clump set. The algorithm now executes once per clump set for each phase—which increases only the coefficients.

4.2.4 Pruning Invalid Solutions

The dependences generated by an application's communication are strictly increasing, so when the algorithm finds that two $\{$*clump, checkpoint location*$\}$ pairs form an invalid recovery line, it has information that it can use to prune other lines without additional validity checks. Recall that an invalid line indicates that a clump, say clump a, is saving fewer events than another clump, say clump b, needs. So for any recovery line where clump a is checkpointing at that checkpoint location, clump b cannot checkpoint at its current location or at any later location. The algorithm can use this information to prune any lines with the clumps at those combinations of checkpoint locations.

4.2.5 Optimizations

Our algorithm is fast and bounds memory usage even though checking recovery line constraints is time consuming and storing recovery lines is memory intensive.

For speed, our algorithm performs only one validity check before generating a line: it compares the first $\{$*clump, checkpoint location*$\}$ pairs in each of the source partial valid recovery lines—if they are compatible, then the line is generated. It also generates many recovery lines for each partition before performing complete validity checks, allowing the algorithm to leverage the constraint information described in Section 4.2.4.

To bound memory usage, though, the algorithm may not generate all of a partition's recovery lines at once. Instead, it generates lines until the number generated reaches a threshold and then prunes the invalid ones. The process of generating and pruning recovery lines is iterative and continues until all lines have been considered.

4.2.6 Branch-and-Bound

To bound memory usage and execution time, the algorithm must limit the number of partial valid recovery lines it keeps from each partition. However, it must also carefully select which lines to discard since it must retain partial valid recovery lines that will result in useful recovery lines. Lines that are less staggered have a better chance of forming valid recovery lines as they are merged to complete lines—the more staggered a line, the more communication happens between each of its checkpoint locations and, thus, the more constraints are introduced and the more likely it is to become invalid as other clumps are introduced.

To find useful recovery lines, the algorithm uses the *branch-and-bound* search strategy [24] in conjunction with our novel *Wide and Flat* (WAF) metric. In this strategy, heuristics tuned to the desired solution—those recovery lines that are both valid and staggered—are used to prune preliminary results. The heuristics are based on the WAF metric, which statically estimates the amount of staggering contained in both partial and complete recovery lines by quantifying the interval of the checkpoint locations (width) and the number of processes that would write their checkpoint at each (flatness). Presently, this metric is a simplified form of the circular string edit distance [25] from the generated line to a perfectly staggered one, and so the goal is to find complete valid recovery lines with low WAF values, which indicate that fewer edits are needed to convert that line to the perfectly staggered line.

At each partition level, the algorithm determines which lines to prune by separating the recovery lines into *bins* based on their WAF values. During pruning, a minimum number of lines is saved in each of the bins, which ensures that some less staggered lines are kept. The details of this policy are discussed in the next section.

4.2.7 Pruning Policy

Our pruning policy is designed to reduce the memory usage and execution time of our algorithm—here we discuss some of the parameters that affect its performance. The settings we use, which are displayed in Table 1, are based on the available memory on the machines we use.

As explained in Section 4.2.4, the algorithm generates some threshold number of recovery lines, the **Line Check Threshold**, before performing validity checks and eliminating invalid lines. A higher number for this threshold results in more lines begin checked for validity at once and thus more potential for the algorithm to leverage information from invalid lines. A higher number also reduces the number of pruning interruptions and decreases the algorithm execution time but causes more lines to be stored, which increases the algorithm's memory usage.

After eliminating the invalid lines, if the number of remaining valid lines is above the **Pruning Threshold**, the algorithm begins deleting some of the latter. The algorithm prunes lines until the total number of lines is some factor of the pruning threshold, **Lines to Keep**. As explained in Section 4.2.6, the algorithm uses each line's WAF value to choose which ones to delete. It leaves a minimum number of lines in each bin (**Minimum Lines in Each Bin**) and deletes the rest; this minimum helps ensure that valid lines are found, since it preserves the lines in higher bins that are more likely to form complete valid lines. This minimum is determined by a formula that approaches zero as the algorithm nears completion.

4.2.8 Correctness Considerations

Some programs delegate communication to wrapper functions, so the context of a function call determines which processes execute the communication. Thus, our algorithm places context-sensitive recovery lines. It does so by inserting a single integer variable in the application text, which is modified to produce a unique value for each context during execution [26]. Process checkpoints are guarded by a condition checking for the appropriate context.

So that process checkpoint calls are placed at locations they execute, our compiler performs a control-dependence analysis. Also, a recovery line must either reside entirely inside or outside of a loop body.

5 Evaluation

We now evaluate our algorithm's scalability, its ability to find useful recovery lines, and the effectiveness of the *Wide and Flat* metric. We also evaluate the performance of the recovery lines identified by our algorithm.

5.1 Implementation

We have implemented our algorithm using the `Broadway` [27] source-to-source ANSI C compiler; it performs context-sensitive interprocedural pointer analysis. Our current implementation simplifies our algorithm. It assumes the application is written in C and uses the MPI communication library. It also assumes that any variables analyzed during the first two phases are regular variables or single-dimension arrays; the variables that are analyzed must not be pointers, multi-dimensional arrays, or loop induction variables. Loops evaluated by our analysis must have a fixed trip count. This implementation assumes that the number of processes the application uses is statically known and that all communication is deterministic.

5.2 Benchmarks

We use three kernel benchmarks and an application benchmark to illustrate the potential for staggered checkpointing. The kernel benchmarks, `BT`, `SP`, and `LU`, are FORTRAN codes from the NAS Parallel Benchmark Suite [6] that we convert to C using `for_c` [28]. `Ek-simple`, our application benchmark, is a well-known computational fluid dynamics benchmark. These benchmarks are indicative of the larger applications in use today.

These benchmarks typically have short execution times, so our experiments use larger than typical problem sizes. We limit the execution to one iteration for our kernel benchmarks to better illustrate how an iteration is affected

Variable	Value	Effect
Pruning Threshold	$\frac{32{,}768}{(sizeofcurrentlines)}$	**increase** to generate more recovery lines and reduce execution time; **decrease** to reduce memory usage
Line Check Threshold	$2048 * (pruningthreshold)$	**increase** to maximize leveraging and reduce execution time; **decrease** to reduce memory
Lines to Keep	$\frac{(pruningthreshold)}{5}$	**increase** to save more lines between pruning passes; **decrease** to reduce pruning frequency and thus time
Minimum Lines in Each Bin	$\frac{Closeness*(linestokeep)}{(numberofbins)}$ where $Closeness < 1$	**increase** to keep more less staggered lines; **decrease** to keep more staggered lines

Table 1. Parameters of the Pruning Policy

by staggered checkpointing. For `Ek-simple`, we report results with 100 iterations but one checkpoint per process.

These applications are simplified—e.g., command-line arguments are set to be constants and function pointers are replaced with static function calls. These simplifications do not affect the overall behavior of the benchmark.

5.3 Algorithm Performance

We report our algorithm's performance including the effects of optimization techniques, which control algorithm complexity and significantly reduce the search space. The compile time necessary to achieve these results is on the order of minutes for 1,024 and 4,096 processes, hours for 16,384 processes, and days for 65,536 processes.

5.3.1 Methodology

We evaluate the algorithm for each of our benchmarks using from 1,024 to 65,536 processes. Since our first two phases, which are not the subject of our research or this paper, scale with the number of processes, we cannot evaluate the algorithm for larger process sizes. For each benchmark configuration, we use a problem size that provides approximately a fifteen minute window for checkpointing.

5.3.2 Results and Analysis

The results of our optimization techniques are presented in Table 2; the column labeled *Initial* defines the size of the starting valid recovery line search space. This space narrows as phases are introduced; we report the size of the search space for a single phase of each benchmark in the *Phases* column. The rest of our results are presented for the same single phase, which is either representative of the phases for that benchmark or is the phase with the majority of the communication.

Our algorithm begins by reducing L, the number of considered checkpoint locations and the base of our search space calculation, by merging checkpoint locations that are too closely spaced to reduce contention. These results are shown in the *Checkpoint Reduction* column of Table 2. For the problem sizes shown in these tables, most checkpoint locations are separated by a sufficient number of local operations to remain individual locations. However, `Ek-simple`'s results show several merged locations.

We also reduce the value of P, the number of considered processes and the exponent of our search space calculation. In the *Clumps* column of Table 2, we report on the results of our clumps technique. Clumps reduce the search space dramatically; the space for `LU` with 1,024 processes is reduced by over 800 orders of magnitude. Clump sizes are relatively even within a configuration, though there is often one clump containing all processes (usually the result of a collective communication call). Also, the number of clumps increases sublinearly with the number of processes, indicating that the clump concept is scalable in practice. Once the clumps are combined into clump sets, as we see in the *Clumps Sets* column, the search space is reduced even further. The number of clump sets is factored out of the exponent of the search space and becomes a multiplier.

Although we have significantly reduced the search space, it is still very large; `LU` with 1,024 processes still has a possible $1.6 * 10^{25}$ recovery lines. However, by leveraging constraint information, first by not propagating invalid constraints in the synthesis algorithm, then by reducing the number of checks required to determine if a recovery line is valid (usually an average of $P^2/2$), and finally by pruning recovery lines less likely to reduce contention, we convert an originally intractable problem into manageable one and enable our algorithm to find useful recovery lines in this space. Table 3 reports the results for each optimization technique employed by our synthesis algorithm. The *Skipped Initially* column represents those lines eliminated by our initial validity check. The next three columns respectively report the number of recovery lines generated, the number eliminated as invalid, and those eliminated as invalid by leveraging the strictly increasing nature of re-

		Possible Recovery Lines				
Benchmark	Processes	Initial	Phases	Checkpoint Reduction	Clumps	Clump Sets
BT	1,024	38^{1024}	14^{1024}	14^{1024}	$5^{96} + 2^{1}$	$3(5^{32}) + 2^{1}$
	4,096	38^{4096}	14^{4096}	14^{4096}	$5^{192} + 2^{1}$	$3(5^{64}) + 2^{1}$
	16,384	38^{16384}	14^{16384}	14^{16384}	$5^{384} + 2^{1}$	$3(5^{128}) + 2^{1}$
	65,536	38^{65536}	14^{65536}	14^{65536}	$5^{768} + 2^{1}$	$3(5^{256}) + 2^{1}$
Ek-simple	1,024	$6^{1024} + 22^{992} + 5^{961} + 12^{31}$	$20^{992} + 4^{961} + 8^{31}$	$13^{992} + 3^{961} + 6^{31}$	$12^{1} + 11^{2} + 9^{31} + 5^{30} + 3^{61}$	$12^{1} + 11^{2} + 10^{1} + 9^{30} + 6^{30}$
	4,096	$6^{4096} + 22^{4032} + 5^{3969} + 12^{63}$	$20^{4032} + 4^{3969} + 8^{63}$	$13^{4032} + 3^{3969} + 6^{63}$	$12^{1} + 11^{2} + 10^{63} + 5^{62} + 3^{125}$	$12^{1} + 11^{2} + 11^{1} + 10^{62} + 6^{62}$
	16,384	$6^{16384} + 22^{16256} + 5^{16129} + 12^{127}$	$20^{16256} + 4^{16129} + 8^{127}$	$13^{16256} + 3^{16129} + 6^{127}$	$12^{1} + 11^{2} + 9^{127} + 5^{126} + 3^{252}$	$12^{1} + 11^{2} + 10^{1} + 9^{126} + 6^{126}$
	65,536	$6^{65536} + 22^{65280} + 5^{65025} + 12^{255}$	$20^{65280} + 4^{16129} + 8^{255}$	$13^{65280} + 3^{65025} + 6^{255}$	$12^{1} + 11^{2} + 9^{255} + 5^{254} + 3^{509}$	$12^{1} + 11^{2} + 10^{1} + 9^{254} + 6^{254}$
LU	1,024	$16^{1024} + 18^{992} + 4^{31}$	$1^{1024} + 8^{992}$	$1^{1024} + 8^{992}$	6^{64}	$2(6^{32})$
	4,096	$16^{4096} + 18^{4032} + 4^{63}$	$1^{4096} + 8^{4032}$	$1^{4096} + 8^{4032}$	6^{128}	$2(6^{64})$
	16,384	$16^{16384} + 18^{16256} + 4^{127}$	$1^{16384} + 8^{16256}$	$1^{16384} + 8^{16256}$	6^{256}	$2(6^{128})$
	65,536	$16^{65536} + 18^{65280} + 4^{255}$	$1^{65536} + 8^{65280}$	$1^{65536} + 8^{65280}$	6^{512}	$2(6^{256})$
SP	1,024	37^{1024}	14^{1024}	14^{1024}	$5^{96} + 1^{1}$	$3(5^{32}) + 1^{1}$
	4,096	37^{4096}	14^{4096}	14^{4096}	$5^{192} + 2^{1}$	$3(5^{64}) + 2^{1}$
	16,384	37^{16384}	14^{16384}	14^{16384}	$5^{384} + 2^{1}$	$3(5^{128}) + 2^{1}$
	65,536	37^{65536}	14^{65536}	14^{65536}	$5^{768} + 2^{1}$	$3(5^{256}) + 2^{1}$

Table 2. Search space reduction by each technique in our algorithm.

covery line constraints. Lastly, the table shows the number of lines pruned by our branch and bound strategy and the number of valid recovery lines produced by our algorithm.

Complexity

The complexity of our algorithm is exponential to the number of clumps in each clump set rather than either 1) the number of clumps in the system or 2) the number of processes in use by the application. Table 2 shows that for most of our benchmarks, the typical number of clumps in a clump set grows with the square root of the number of processes. `Ek-simple` shows an even smaller growth rate.

Wide and Flat Metric

Our WAF metric separates useful recovery lines from other generated lines for complete and partial recovery lines.

By comparing lines with different WAF values for several configurations of each benchmark, we found that the WAF metric performs well particularly for configurations with large numbers processes and amounts of checkpoint data. Table 4 shows a snapshot of these results.

While for smaller configurations the WAF metric always identifies a line faster than the line in which every process writes its checkpoint at approximately the same time, the line with the lowest WAF value may not be the fastest. This situation happens with smaller amounts of checkpoint data because recovery lines that are less staggered can sufficiently reduce network and file system contention to maintain system performance—and less separation of checkpoints causes less disruption of communication, which is an overall benefit to the application.

5.4 Performance of the Identified Lines

Our results show that staggered recovery lines identified by our algorithm reduce checkpoint time when *simultaneous*

		Recovery Lines					
Benchmark	**Processes**	**Skipped Initially**	**Generated**	**Investigated**	**Leveraged**	**Pruned by WAF**	**Valid Found**
BT	1,024	18,606	4,603,597	4,603,597	0	4,588,183	620
	4,096	50,262	9,325,092	9,325,092	0	9,293,840	432
	16,384	77,206	18,731,170	18,731,170	0	18,668,236	442
	65,536	156,414	37,759,736	37,759,736	0	37,632,812	408
Ek-simple	1,024	401,967	2,465,869	924,627	1,541,242	193,270	408
	4,096	1,187,522	8,243,283	2,850,232	5,393,051	2,819,754	204
	16,384	3,598,557	10,667,118	3,620,224	7,046,894	3,571,133	102
	65,536	3,889,674	22,989,972	8,860,009	14,129,963	8,763,342	50
LU	1,024	1,056	3,233,318	2,615,253	618,065	2,603,302	409
	4,096	2,046	6,603,040	5,604,553	998,487	5,580,103	203
	16,384	128,655	13,129,553	10,591,018	2,538,535	10,542,142	103
	65,536	8,382	26,464,194	19,406,505	7,057,689	19,308,542	191
SP	1,024	22,573	4,600,329	4,600,329	0	4,584,915	614
	4,096	45,352	9,331,318	9,331,318	0	9 299,935	538
	16,384	91,860	18,927,310	18,927,310	0	18,863,945	188
	65,536	179,544	39,437,768	39,437,768	0	39,308,070	169

Table 3. Pruning performed by our algorithm.

checkpointing, which occurs when all processes write their checkpoints at the same location in the application text, saturates the file system. However, when a sufficiently small number of processes writes a sufficiently small amount of data such that the file system is not saturated, simultaneous checkpointing is better than staggered checkpointing, since the latter can disrupt communication.

In our analysis, we indicate configurations where the lines identified by our algorithm are in the *sweet spot* for staggered checkpointing, or where: 1) staggered checkpointing reduces checkpoint time by at least 25% and to less than fifteen minutes, and 2) simultaneous checkpointing requires more than five minutes.

5.4.1 Methodology

To evaluate the recovery lines identified by our algorithm, we compare a line with a low WAF value, which is a staggered line, against a line with a high WAF value. The lines with high WAF values typically perform simultaneous checkpointing. We present results showing the decrease in checkpointing time achieved by staggered checkpointing on Ranger [29], a supercomputer located at the Texas Advanced Computing Center [30]. Ranger consists of local hardware with four 2.3 GHz processors with four cores each, an Infiniband network, and a Lustre file system featuring 40 GB/s throughput. The results presented here are from simulation due to usage restrictions on the production systems to which we have access. We used our simulator [22] , which was validated against Ranger [29], and on average predicts the execution time of our benchmarks with checkpointing as 83% of their actual measured performance. This accuracy suffices to evaluate our solution—we show that it is effective using confidence (95%) and tolerance (95%) intervals.

Our simulator performs optimistically in the face of file system saturation, so we do not present results for configurations that cause both forms of checkpointing to saturate the file system. In addition, unlike our algorithm, our simulator scales with the number of processes used by the application. As such, it cannot simulate in a reasonable amount of time two of our benchmarks with 16,384 processes or any of the benchmarks with 65,536 processes. Thus, those results are not presented.

To better analyze our results, we have fixed the checkpoint size for each configuration.

5.4.2 Results and Analysis

Tables 5 and 6 present the decreases in checkpointing time realized by the staggered line over the simultaneous line: this data supports both our claim that staggered checkpointing improves checkpointing performance and our claim that the WAF metric identifies useful recovery lines. The decrease is adjusted for the average error of our simulator, and these tables also display the 95% confidence interval around the mean and the tolerance interval indicating the range for values of 95% of the population with 95% certainty. Table 5 shows the results for when the checkpoint locations are staggered within a three minute window; Table 6 shows the results for when the window is increased to fifteen minutes. These time periods translate to three or fifteen minutes without a synchronization point or a collective communication, respectively. In today's applications, three minutes is closest to the expected interval, though the intervals are often smaller. Fifteen minutes represents the amount of time we consider reasonable for an application to spend checkpointing. In both of these tables, configura-

	Checkpoint Time	
WAF value	4 MB	32 MB
418	12m	67m
3,958	4m	78m
7,008	18m	93m
7,361	19m	94m

Table 4. The simulated checkpoint times for recovery lines.

tions falling into the sweet spot are in bold.

When checkpoint locations are separated within an approximately three minute window, the staggered lines placed by our algorithm reduce checkpoint time by an average of 24% (Table 5). Additionally, our algorithm finds lines that bring three configurations into the sweet spot. We conclude that the WAF metric can identify useful recovery lines even within this small interval size.

If we increase the window size to approximately fifteen minutes (Table 6), checkpoint time typically decreases by an even larger amount; the average decrease is now 44%. The increased window size also brings two more configurations into the sweet spot, again pointing to the effect of both staggered checkpointing and our WAF metric.

6 Conclusions and Future Work

We designed and implemented a new compile-time algorithm that generates and places useful recovery lines in applications that use up to tens of thousands of processes. This algorithm uses a constraint-based search to eliminate redundant work and reduces the search space by constraining the checkpoint locations, considering clumps of processes rather than independent processes, and performing the search for useful recovery lines within sets of clumps where each process is represented exactly once. These techniques reduce the search space for BT with 1,024 processes from 38^{1024} to $3(5^{32}) + 2^1$, or by 1,594 orders of magnitude. We consider the algorithm scalable since it causes the search space to grow much more slowly than the number of processes: typically the space grows with the square root of the number of processes.

Our pruning policy enables the rapid sorting of the created recovery lines, and the metric we introduced, Wide and Flat (WAF), statically estimates the usefulness of both complete and partial recovery lines.

For our benchmarks, our implementation finds and places useful recovery lines in applications that use up to 65,536 processes. The staggered recovery lines placed by our algorithm checkpoint an average of 35% faster than simultaneous checkpointing.

In the future, we plan to improve the lines identified by our algorithm by adjusting the checkpoints in the generated recovery lines relative to the checkpoint locations we consider. Such adjustments may both increase the separation of the checkpoints and reduce the amount of checkpoint data. We also plan to build a recovery system to complement our checkpoint methodology.

7 Acknowledgments

The authors would like to acknowledge Sung-Eun Choi of Cray, Inc. for her technical contributions to this work. In addition, the authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported in this paper.

References

[1] Y. Liang et al. Filtering failure logs for a Blue Gene/L prototype. In *DSN*, 2005.

[2] Y. Liang et al. Blue Gene/L failure analysis and prediction models. In *DSN*, 2006.

[3] B. Schroeder et al. A large-scale study of failures in high-performance computing systems. In *DSN*, 2006.

[4] Ö. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. Technical Report UBLCS-93-1, Laboratory for Computer Science, University of Bologna, Italy, 1993.

[5] A. N. Norman, S.-E. Choi, and C. Lin. Compiler-generated staggered checkpointing. In *Workshop on Languages, Compilers, and Runtime Support for Scalable Systems*, 2004.

[6] NASA Ames Research Center. NAS parallel benchmarks. `http://www.nas.nasa.gov/Software/NPB`.

[7] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (BLCR) for linux clusters. *Journal of Physics: Conference Series*, 40:494–499, 2006.

[8] J. P. Walters et al. *High Performance Computing*, volume 4873/2007 of *LNCS*, chapter A Scalable Asynchronous Replication-Based Strategy for Fault Tolerant MPI Applications, pages 257–268. Springer Berlin / Heidelberg, 2007.

[9] A. Moody et al. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Supercomputing*, 2010.

[10] J. F. Ruscio et al. DejaVu: Transparent user-level checkpointing, migration and recovery for distributed systems. In *Supercomputing*, 2006.

[11] J. S. Plank. *Efficient checkpointing on MIMD architectures*. PhD thesis, Princeton University, Princeton, NJ, USA, 1993.

[12] N. H. Vaidya. Staggered consistent checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):694–702, 1999.

[13] N.P. Gopalan and K. Nagarajan. *Distributed Computing—IWDC*, volume 3741/2005 of *LNCS*, chapter Self-refined Fault Tolerance in HPC Using Dynamic Dependent Process Groups, pages 153–158. Springer Berlin / Heidelberg, 2005.

[14] J. C. Y. Ho et al. Scalable group-based checkpoint/restart for large-scale message-passing systems. In *IPDPS*, 2008.

[15] I. R. Philp. Software failures and the road to a petaflop machine. In *HPCRI*, 2005.

[16] A. Guermouche et al. Uncoordinated checkpointing without domino effect for send-deterministic message passing applications. In *IPDPS*, 2011.

[17] S.-E. Choi and S. J. Deitz. Compiler support for automatic checkpointing. In *HPCA*, 2002.

[18] G. Rodríguez et al. CPPC: a compiler-assisted tool for

Benchmark	Processes	Checkpoint Data Per Process	Decrease in Checkpoint Time Adjusted for Average Error		95% Confidence Interval	Tolerance Interval
			Sec	(%)		
BT	4,096	4MB	**430s**	**(39%)**	35%-42%	34%-43%
		32MB	1,365s	(25%)	21%-29%s	20%-30%
	16,384	4MB	8,952s	(52%)	49%-55%	48%-56%
Ek-simple	4,096	4MB	**493s**	**(45%)**	41%-48%	40%-49%
		32MB	**2,342s**	**(43%)**	40%-46%	39%-47%
LU	4,096	4MB	222s	(20%)	16%-24%	15%-25%
		32MB	207s	(4%)	0%-8%	-1%-9%
SP	4,096	4MB	94s	(9%)	5%-12%	3%-14%
		32MB	84s	(2%)	-3%-6%	-4%-7%
	16,384	4MB	74s	(0%)	-4%-5%	-5%-6%

Table 5. Decrease in checkpointing time when checkpoints are staggered within an approximately three minute window.

Benchmark	Processes	Checkpoint Data Per Process	Decrease in Checkpoint Time Adjusted for Average Error		95% Confidence Interval	Tolerance Interval
			Sec	(%)		
BT	4,096	4MB	**381s**	**(34%)**	31%-38%	30%-39%
		32MB	1,153s	(21%)	17%-25%s	16%-26%
	16,384	4MB	9,064s	(52%)	49%-56%	48%-57%
Ek-simple	4,096	4MB	**1,071s**	**(97%)**	94%-100%	93%-100%
		32MB	**5,137s**	**(94%)**	91%-97%	91%-98%
LU	4,096	4MB	**788s**	**(71%)**	68%-74%	67%-75%
		32MB	876s	(16%)	12%-20%	11%-21%
SP	4,096	4MB	**616s**	**(56%)**	52%-59%	51%-60%
		32MB	556s	(10%)	6%-14%	5%-15%
	16,384	4MB	814s	(5%)	1%-9%	0%-10%

Table 6. Decrease in checkpointing time when checkpoints are staggered within an approximately fifteen minute window.

portable checkpointing of message-passing applications. *Concurrency and Computation*, 22:749–766, 2010.

[19] G. Bronevetsky et al. Automated application-level checkpointing of MPI programs. In *PPoPP*, 2003.

[20] A. Agbaria et al. Application-driven coordination-free distributed checkpointing. In *ICDCS*, 2005.

[21] E. N. Elnozahy et al. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[22] A. N. Norman. *Compiler-Assisted Staggered Checkpointing*. PhD thesis, The University of Texas at Austin, 2010.

[23] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, Inc., San Diego, California, 1993.

[24] S. Beale. *Hunter-Gatherer*. PhD thesis, Carnegie Mellon University, 1997.

[25] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. The Press Syndicate of The University of Cambridge, 1997.

[26] B. Wiedermann. Know your place: Selectively executing statements based on context. Technical Report TR-07-38, UT Austin, 2007.

[27] Samuel Z. Guyer and Calvin Lin. Broadway: A software architecture for scientific computing. In R.F. Boisvert and P.T. P. Tang, editors, *The Architecture of Scientific Software*. Kluwer Academic Press, 2000.

[28] FOR_C. http://www.cobalt-blue.com, 1988.

[29] Texas Advanced Computing Center. *Ranger User Guide*.

[30] Texas Advanced Computing Center (TACC). Austin, TX.